DOI: http://dx.doi.org/10.29069/forscience.2014v2n1.e54

PORTANDO UMA APLICAÇÃO DE SISTEMA EMBARCADO COM ARQUITETURA SUPER LOOP PARA OPERAR COM SISTEMA OPERACIONAL DE TEMPO REAL

Edielson Prevato Frigieri[1]
Vitor Ivan D'Angelo[2]
Rômulo Franklin de Meira Ramos[3]
Luiz Cláudio de Andrade Jr.[4]
Pavel Suene Félix Lopes Teixeira[5]

RESUMO

Atualmente, a utilização dos sistemas operacionais, principalmente os sistemas de tempo real (RTOS), tem sido de grande importância no desenvolvimento de sistemas embarcados. Cada vez mais, diferentes e variadas funcionalidades são requisitos dos projetos de sistemas embarcados, como interfaces de comunicação com e sem fio, interfaces com o usuário através de displays, interfaces de acesso de dado como USB, serial, etc. Com o aumento da complexidade, fica extremamente difícil construir um sistema que seja estável e que atenda aos requisitos de tempo a partir de estruturas de programas comuns como o *Super Loop*. A fim de demonstrar os problemas inerentes deste tipo de arquitetura, foi desenvolvida uma aplicação utilizando o *Super Loop* e a mesma foi portada para uma solução baseada em RTOS demonstrando os principais conceitos que envolvem um sistema operacional, como utilizá-lo, e os benefícios obtidos com a sua utilização, como estabilidade do sistema, sincronização entre tarefas e cumprimento dos requisitos de tempo.

Palavras-chave: Sistemas operacionais de tempo real. Sistemas embarcados. Escalonamento.

PORTING AN EMBEDDED SYSTEM APPLICATION WITH SUPER LOOP ARCHITECTURE TO OPERATE WITH REAL-TIME OPERATING SYSTEM

ABSTRACT

Currently, the use of operating systems, especially real-time systems (RTOS), has been of great importance in the development of embedded systems. Increasingly, different and varied features are requirements for embedded system projects, as wired and wireless communication interfaces; user interfaces through displays; data access interfaces such as USB, serial, etc. With the increase in complexity, it becomes extremely difficult to build a system that is stable and that meets the time requirements from common program structures like Super Loop. In order to demonstrate the inherent problems of this type of architecture, an application was developed using Super Loop and it was ported to a solution based on RTOS, demonstrating the key

[1] Doutorando em Engenharia de Produção pela Universidade Federal de Itajubá (Unifei) e professor auxiliar do Instituto Nacional de Telecomunicações (Inatel). E-mail: edielson@inatel.br.
[2] Bacharelando em Engenharia da Computação pelo Inatel. E-mail: vitor.dangelo@gec.inatel.br.
[3] Bacharelando em Engenharia de Telecomunicações pelo Inatel. E-mail: romulo.ramos@inatel.br.
[4] Bacharelando em Engenharia da Computação pelo Inatel. E-mail: luiz.junior@gec.inatel.br.
[5] Bacharelando em Engenharia da Computação pelo Inatel. E-mail: pavel.suene@gec.inatel.br.

concepts involving an operating system, how to use it and the benefits gained from its use as system stability, synchronization between tasks and fulfillment of time requirements.

Keywords: Real-time operating systems. Embedded systems. Scheduler.

1 INTRODUÇÃO

Atualmente, o número de processadores utilizados em sistemas embarcados ultrapassa o de processadores em computadores pessoais *Personal Computers* (PC), o que mostra a importância do seu mercado dentro da área de sistemas computacionais (MARWEDEL, 2011). Um exemplo disto é que alguns carros de última linha chegam a ter mais de 100 processadores. Estes números são bem maiores do que o esperado, visto que a grande maioria das pessoas não percebe que está utilizando algum tipo de processador em seus produtos ou aparelhos no dia a dia.

Dentro da área de desenvolvimento de software embarcado, é muito comum a utilização de algum tipo de Sistema Operacional (SO). No Gráfico 1 é apresentada uma pesquisa de mercado para sistemas embarcados, realizada pela CMP, EE Times em 2013, com o percentual de usuários que utilizam algum tipo de SO em seu projeto. O resultado mostra que, apesar de uma pequena diminuição nos últimos cinco anos, 68% dos projetos utilizam algum tipo de sistema operacional.

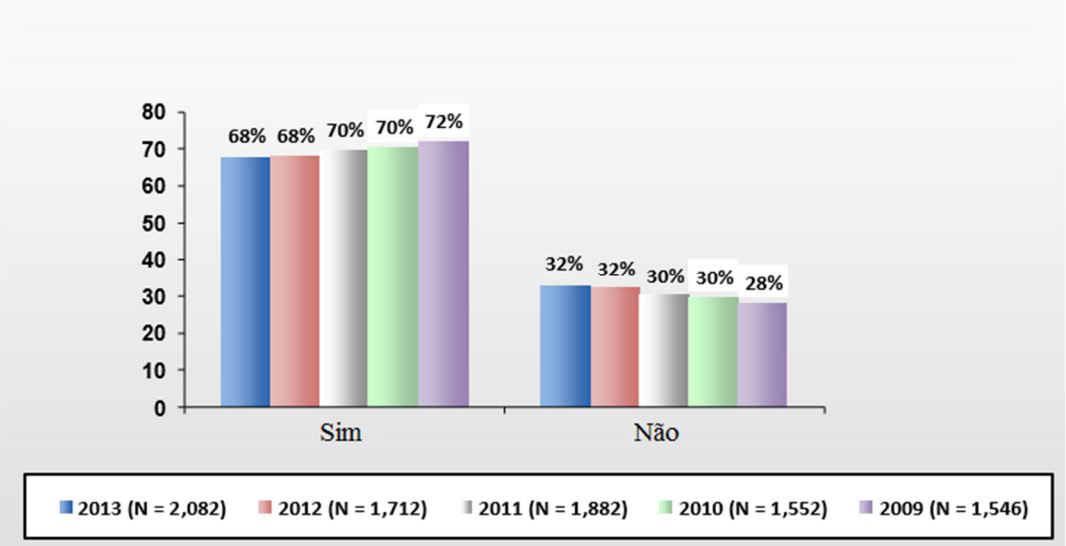

Gráfico 1 - Utilização de SO
Fonte: Embedded market survey (MEDIA, 2013).

Existem diferentes tipos de sistemas computacionais com propósitos específicos. Dentre eles, estão os sistemas operacionais de tempo real (RTOS), desenvolvidos para sistemas

embarcados com rígidos requisitos de tempo, como, por exemplo, os sistemas utilizados em automóveis, robôs e alguns dispositivos de consumo eletrônicos.

Para este tipo de sistema, o processamento deve ser realizado dentro de um tempo fixo, ou o sistema falhará (SILBERSCHATZ; GALVIN; GAGNE, 2012).

Para entender o que é um sistema operacional de tempo real, é necessário primeiramente definir sistema operacional, que é um pacote de software que gerencia os recursos de hardware de um sistema, facilitando o desenvolvimento de aplicações neste (MOHAMADI, 2011). A diferença entre um RTOS e um SO genérico é que o primeiro é desenvolvido para que o escalonamento atinja respostas em tempo real (ANH; TAN, 2009), ou seja, sistemas embarcados de tempo real precisam ser capazes de responder e tratar eventos do sistema dentro de uma limitação de tempo predefinida.

O crescimento da utilização deste tipo de sistema operacional no desenvolvimento de aplicações embarcadas motivou a realização deste trabalho. Conforme pode ser visto no Gráfico 2, um dos requisitos mais buscados nos projetos atuais é a resposta em tempo real. Além disso, sistemas operacionais de tempo real podem auxiliar na resolução de alguns problemas recorrentes em programação de microcontroladores e microprocessadores, como a necessidade de executar múltiplas tarefas de forma concorrente, gerenciar as prioridades das tarefas que serão executadas, sincronizar eventos de interrupção com o restante do sistema, gerenciar memória e outros recursos de hardware, dentre outros (DOLINAY; VAŠEK; DOSTÁLEK, 2011). Além disso, podem trazer facilidades, como a reutilização de código - que reduz o tempo de desenvolvimento - e maior modularização da solução, beneficiando o trabalho em equipe.

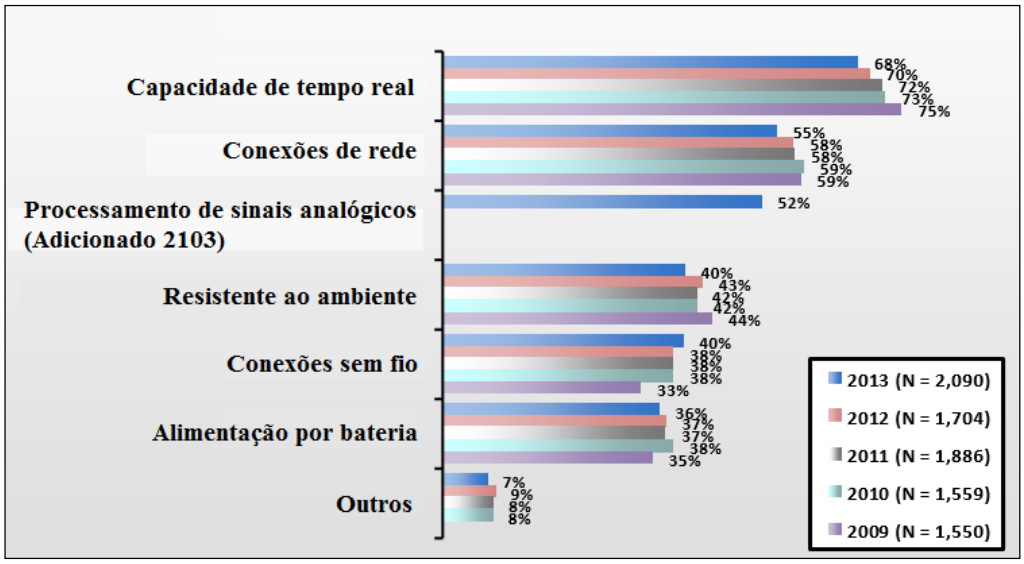

Gráfico 2 – Requesitos de projeto
Fonte: Embedded market survey (MEDIA, 2013).

Com base no crescimento do mercado de sistemas embarcados e na necessidade de atender a requisitos de tempo através da utilização de sistemas operacionais, este trabalho apresenta os benefícios da utilização de um RTOS, assim como os passos para migrar de uma solução baseada em *Super Loop* (SL) para um sistema com SO. Para exemplificar os passos de migração, será adotado o FreeRTOS™ - um sistema operacional com características que o qualificam para este trabalho, conforme será detalhado nas seções posteriores.

Este artigo está organizado da seguinte forma: na Seção 2, será apresentado o tipo de solução comumente utilizada em sistemas embarcados, conhecida como *Super Loop*, assim como suas vantagens e desvantagens; na Seção 3, serão discutidas as características de um RTOS e o que o torna uma ferramenta eficiente para projetos com requisitos de resposta em tempo real; na Seção 4, apresentar-se-á o FreeRTOS™ e suas principais características; os passos para portar um projeto de um sistema SL para um sistema com RTOS, no caso o FreeRTOS™, serão descritos na Seção 5; por fim, o trabalho será concluído na Seção 6.

2 SISTEMAS SUPER LOOP

Sistemas SL, também conhecidos como sistemas *foreground/background*, são constituídos de um loop infinito (*background*) que faz as chamadas das tarefas de modo sequencial, onde uma tarefa inicia após o término da anterior. A esta forma de chaveamento entre tarefas dá-se o nome de escalonamento cooperativo (NAHAS, 2011).

O tratamento de interrupções *Interrupt Service Routine* (ISR), que pode ocorrer de forma assíncrona no sistema, é chamado de *foreground*. Uma melhor ilustração sobre o método SL pode ser vista no Gráfico 3.

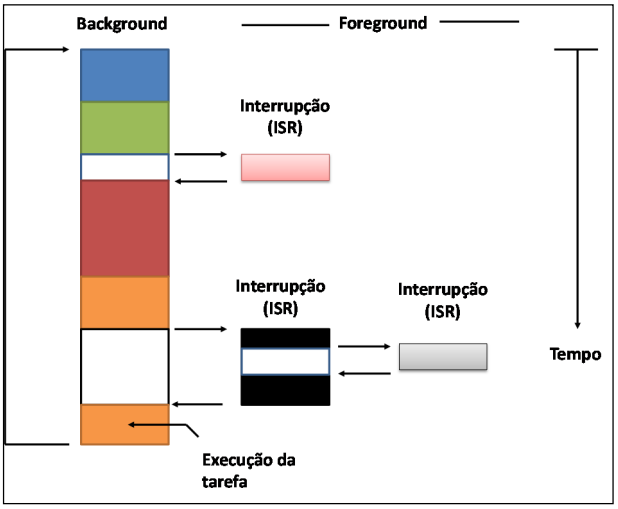

Gráfico 2 - Exemplo de sistema *foreground/background*.
Fonte: Autor.

2.1 Vantagens do Super Loop

Devido à simplicidade e facilidade de implementação, o SL é uma das soluções mais utilizadas em projetos de sistemas embarcados. Uma demonstração de como é codificada esta solução é ilustrada na Figura 1. Como o escalonamento para este tipo de sistema depende do hardware, geralmente apresenta bom tempo de resposta, quando se tem um número conhecido e fixo de tarefas.

```
void main( void )
{
       init(); /* inicialização do sistema. */

       /* implementado como um loop infinito. */
       while( TRUE )
       {
           /* Chamadas das tarefas */
           Tarefa_1();
           Tarefa_2();
           ...
           Tarefa_N();
       }
}
```

Figura 1 - Código para sistema baseado em *super loop*
Fonte: Autor

2.2 Desvantagens do Super Loop

Por outro lado, se o número de tarefas do sistema aumentar consideravelmente, passa a ser difícil controlar o tempo de resposta do sistema, tornando-o vulnerável a variações de tempo e a falhas de hardware (LAPLANTE; OVASKA, 2011). Ou seja, qualquer demora no tratamento de alguma tarefa ou de uma interrupção (*foreground*) pode trazer atrasos ou até mesmo falhas no sistema. Dentre as principais desvantagens do *Super Loop,* podem-se destacar:

a) dificuldade em garantir que uma tarefa seja executada dentro de uma restrição de tempo;
b) todas as tarefas do background possuem a mesma prioridade;
c) se houver algum atraso em uma das tarefas, todo o sistema será impactado;
d) atrasos nas tarefas de ISR (*foreground*) também podem gerar atrasos em todo o sistema;
e) dificuldade de sincronização de tarefas em *background* e *foreground*.

Logo, para soluções em que o tempo de resposta das tarefas é um requisito, faz-se necessário utilizar alguma forma de gerenciamento que garanta o cumprimento dos requisitos. Uma das soluções é a utilização de um sistema operacional de tempo real.

A próxima seção apresentará as vantagens e desvantagens de se utilizar um RTOS.

3 SISTEMAS OPERACIONAIS DE TEMPO REAL

Sistema operacional de tempo real é um subtipo de sistema operacional que possui a maioria das características de um SO convencional. É responsável por controlar e gerenciar os vários recursos de hardware e disponibilizar o acesso a estes recursos através de chamadas do sistema. Porém, as principais propriedades que diferenciam um RTOS de um SO convencional são as seguintes:

a) **Características de tempo real:** representam a precisão do tempo de resposta. Toda tarefa do sistema precisa ter um tempo determinado e ser atendida dentro deste tempo, a fim de garantir uma resposta a eventos em um prazo limitado (MOHAMADI, 2011; SHEN *et al.*, 2009). Esta pode ser considerada uma das características-chave que diferem um RTOS de um SO genérico. No RTOS, o tempo de expedição de uma tarefa, a latência da alternância entre tarefas e a latência de interrupção devem ter um tempo previsível e consistente, mesmo quando o número de tarefas aumenta (ANH; TAN, 2009). Basicamente, a característica de tempo real pode ser dividida entre *hard real-time* e *soft real-time* (SHEN *et al.*, 2009). Em sistemas *hard real-time*, os eventos internos e externos devem ser atendidos em tempo predeterminado e as tarefas devem ser finalizadas dentro do prazo. O sistema de abertura de *airbag* é um ótimo exemplo de sistema *hard real-time*, onde não pode haver nenhum tipo de atraso na resposta do sistema. *Soft real-time* significa que o sistema não tem restrições de tempo tão rígidas quanto o *hard real-time* e, portanto, um evento pode sofrer pequenas variações de tempo a cada execução (SILBERSCHATZ; GALVIN; GAGNE, 2012).
b) **Escalonamento com prioridade:** para os sistemas operacionais de uso geral, o escalonamento tem o objetivo de distribuir igualmente os recursos do sistema entre as tarefas da forma mais justa possível. Quase sempre são desenvolvidos utilizando-se algoritmos *round-robin* para o escalonamento de tarefas, com base na divisão do tempo total de processamento disponível (*time slice*), conforme demonstrado Figura 2. Já para o caso de sistemas operacionais de tempo real, em alguns momentos é necessário dar

total atenção a alguma tarefa crítica, a fim de garantir que ela seja executada. Para que isto ocorra, o método de escalonamento de tarefas deve ser do tipo preemptivo, onde é possível definir prioridades para as tarefas do sistema, permitindo que aquelas de maior prioridade interrompam incondicionalmente as menos prioritárias, para utilizar os recursos de CPU necessários (ANH; TAN, 2009).

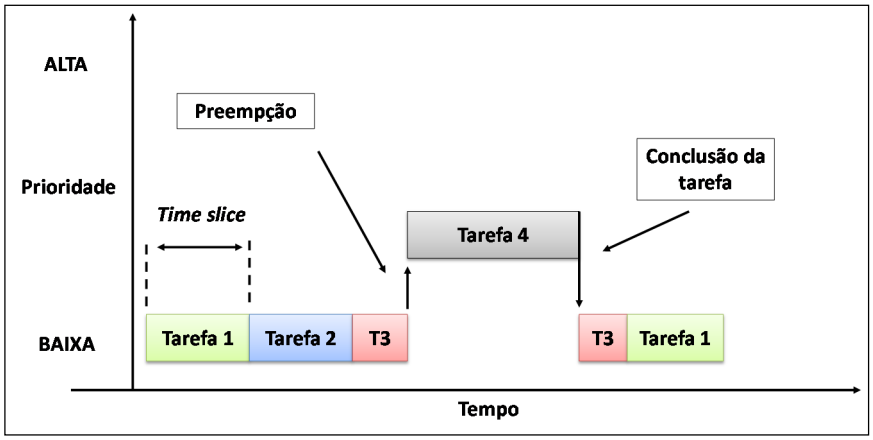

Figura 2 - Exemplo de escalonamento preemptivo com prioridade
Fonte: Autor.

c) **Compartilhamento de dados**: normalmente, sistemas embarcados não possuem memória virtual. Dessa forma, todas as tarefas do sistema compartilham a mesma região de memória, o que pode causar algum tipo de conflito de acesso. Para resolver este problema, alguma forma de gerenciamento de memória precisa ser utilizada. Porém, este gerenciamento consome recursos e reduz o desempenho do sistema operacional. Uma forma eficiente de gerenciar a memória é através de mecanismos de *mutex* (*mutual exclusion*), que conseguem evitar conflitos de acesso, simplificando as operações de leitura e escrita por não precisarem de mapeamentos de memória mais complicados ou da utilização de pipelines (SHEN *et al.*, 2009).

d) **Previsibilidade na sincronização de tarefas**: a sincronização entre tarefas em um RTOS deve acontecer com tempo previsível, utilizando alguma ferramenta para essa finalidade, como semáforo, filas de mensagem, *flag* de evento etc. Em SO genéricos, não é possível sincronizar tarefas porque o próprio SO introduz atrasos no sistema (ANH; TAN, 2009).

Como desvantagem, os serviços providos adicionam um *overhead* de execução, que pode variar entre 2% e 5% do uso da CPU, dependendo do RTOS. Um espaço extra de memória ROM ou FLASH é necessário para armazenar o código, que pode ser de algumas centenas de bytes até algumas centenas de milhares de bytes. Além da memória de programa, um RTOS

consome RAM para o armazenamento do contexto e da *stack* de cada tarefa, que varia entre algumas centenas de bytes até algumas dezenas de milhares de bytes.

Para este trabalho, escolheu-se um RTOS que permita a realização da análise dos benefícios da utilização de um sistema operacional em um projeto de sistema embarcado. O RTOS escolhido foi o FreeRTOS™, que terá suas principais características detalhadas na próxima seção.

4 FREERTOS™

De acordo com BARRY (2010), o FreeRTOS™ é um *kernel* de tempo real, ou um escalonador de tempo real, que pode ser utilizado em aplicações onde os requisitos de tempo são realmente críticos. Ele permite que as aplicações sejam organizadas como uma coleção de *threads* independentes. Como em sistemas embarcados comumente são utilizados processadores de somente um core, apenas uma tarefa pode ser executada por vez, logo, o *kernel* pode decidir qual tarefa será executada, dependendo da prioridade de cada uma delas. Dessa forma, o desenvolvedor pode associar altas prioridades às tarefas com características *hard real-time* e prioridades menores para aquelas com características *soft real-time*.

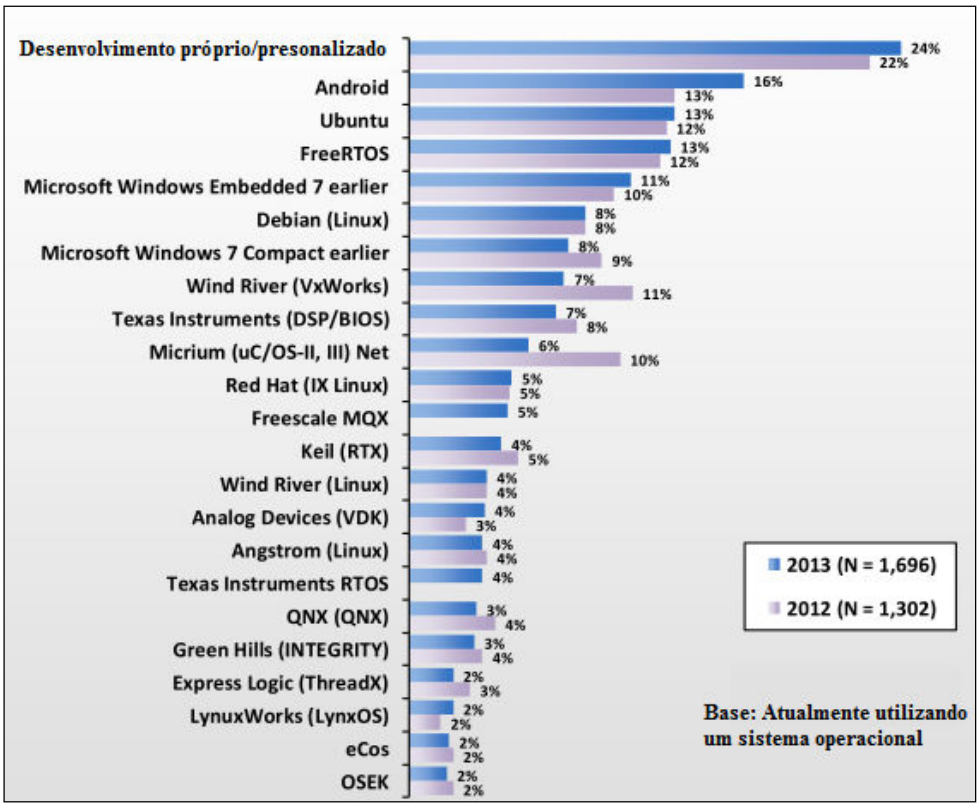

Gráfico 4 - Distribuição do uso de SO
Fonte: Embedded market survey (MEDIA, 2013).

Este sistema operacional foi escolhido para o presente trabalho primeiramente por ser de código aberto, o que facilita o estudo de seu funcionamento. Em segundo lugar, por ser um dos mais utilizados atualmente, conforme pode ser visto no Gráfico 4.

Por ser um *kernel* de tamanho pequeno (em torno de 6 kB de flash e algumas centenas de bytes de RAM) e facilmente portável para qualquer arquitetura, o FreeRTOS™ tem conquistado o mercado de sistemas embarcados para pequenos microcontroladores. Algumas características deste SO que podem ser destacadas são:

a) **Tarefas como pequenos programas**: cada tarefa deve ser criada como um pequeno programa que contém um ponto de entrada e que roda infinitamente, sem chegar ao fim de sua execução, ou seja, não retornará nunca à função chamadora (BARRY, 2010). Um exemplo de como ficaria o código de uma tarefa utilizando o FreeRTOS™ pode ser visto na Figura 3.

```
void vExemploDeTarefa( void *pvParametros )
{
        int iExemploDeVariavel=0;

        /* implemenado como um loop infinito. */
        for( ;; )
        {
            /* O código da tarefa deve
               ser implementado aqui */
        }

        /* Se por algum motivo a tarefa sair do
        loop, a mesma deve ser deletada*/
        vTaskDelete(NULL);
}
```

Figura 3 - Protótipo de tarefa para o FreeRTOS™
Fonte: Autor.

b) **Escalonamento preemptivo**: como cada tarefa roda infinitamente, ela só pode liberar recursos do processador para outra tarefa se for interrompida. Por esta razão, o FreeRTOS™ possui escalonamento preemptivo, o que permite que uma tarefa com maior prioridade interrompa uma de prioridade mais baixa durante sua execução, tomando os recursos do sistema e possibilitando o atendimento incondicional à tarefa de maior prioridade (BARRY, 2010).

c) **Gerenciamento de tarefas**: uma aplicação pode ser constituída de várias tarefas. Como apenas uma tarefa pode ser executada de cada vez (considerando um processador de

apenas um núcleo), o sistema operacional é responsável por salvar o contexto de execução da tarefa atual para então carregar outra, ou seja, salvar os valores atuais armazenados nos registradores da CPU e carregar os valores da nova tarefa a ser executada (HASAN; AHMAD, 2008). A esta ação dá-se o nome de troca de contexto (*context switching*), que pode ser executada somente pelo escalonador (*kernel*). Esta operação está ilustrada na Figura 4.

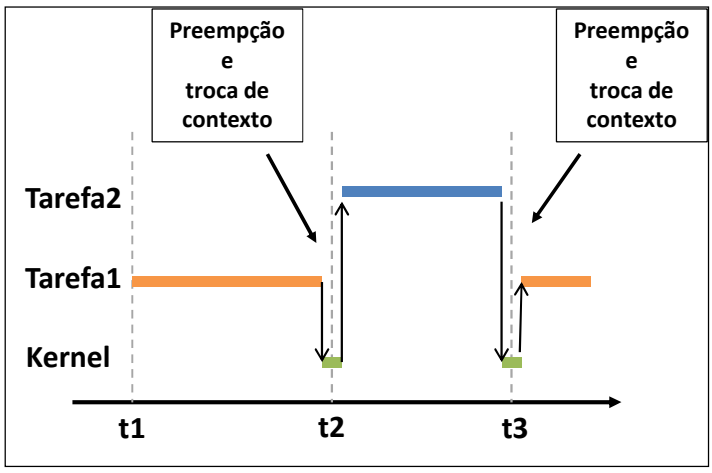

Figura 4 - Exemplo de troca de contexto executada pelo *kernel*
Fonte: Autor

d) **Gerenciamento de filas**: as aplicações que utilizam o FreeRTOS™ são estruturadas como um conjunto de tarefas independentes. Apesar de cada tarefa ser um pequeno programa autônomo, muitas vezes é necessário trocar informações entre elas. Para solucionar este requisito, foi criado um sistema baseado em estruturas de fila (*queue*). Estas estruturas podem armazenar uma quantidade finita de itens com valores fixos e devem ser utilizadas com o tipo de acesso FIFO (*First in First Out*). As filas são objetos do sistema que não fazem parte de uma tarefa específica. Portanto, os dados inseridos nestas estruturas podem ser acessados por qualquer tarefa, permitindo a troca de dados e a sincronização do sistema (BARRY, 2010).

e) **Gerenciamento de interrupções**: sistemas embarcados de tempo real precisam tratar eventos externos que venham a ocorrer através de interrupções. Pode-se citar como exemplo a chegada de um pacote em uma interface Ethernet, que precisa ser encaminhado para a pilha TCP/IP para que seja processado. Sistemas mais complexos terão que tratar eventos com origens diversas, cada um com um requisito de tempo de resposta. O FreeRTOS™ possui ferramentas que facilitam a sincronização de

interrupções com o restante do sistema, seja qual for a estratégia de tratamento adotada pelo desenvolvedor.

f) **Gerenciamento de memória**: cada sistema embarcado possui requisitos específicos para tempo e alocação de memória RAM. O FreeRTOS™ trata a alocação dinâmica de memória como parte da camada de portabilidade do sistema, permitindo que as aplicações possam utilizar sua própria implementação, quando apropriado (BARRY, 2010). Para pequenos sistemas embarcados, é comum a criação de tarefas, filas e semáforos, antes da inicialização do escalonador. Dessa forma, a memória só é dinamicamente alocada pelo *kernel* antes da aplicação começar qualquer operação em tempo real e continua alocada por todo o tempo de execução da aplicação. Isto implica que não é necessário considerar qualquer esquema complexo de alocação de memória como fragmentação ou determinismo.

5 PORTANDO UMA APLICAÇÃO PARA O RTOS

Para exemplificar a migração de uma aplicação de um sistema baseado em *Super Loop* para um sistema baseado em RTOS, será utilizado o processador NXP LPC1769, que é um microcontrolador fundamentado na arquitetura ARM Cortex-M3 para aplicações embarcadas, que opera em uma frequência de até 120 MHz, até 512 kB de memória flash e até 64 kB de memória de dado. Os periféricos que serão acessados pelo processador (entrada analógica, display OLED etc.), no exemplo mostrado a seguir, fazem parte da placa LPCXpresso Base Board fornecido pela Embedded Artists.

Um código simples desenvolvido com base na arquitetura SL, ilustrado na Figura 5, será a referência utilizada para a portabilidade. Antes de iniciar o processo de migração, é necessário entender o que faz esse exemplo.

Essa aplicação pode ser dividida em duas funcionalidades principais. Primeiramente, tem-se uma leitura de dados que é realizada através de um conversor analógico digital (AD), pela chamada da função "**ADC_ChannelGetData**". Vale ressaltar que, antes desta leitura, existe uma estrutura de repetição ***while*** que só é terminada se a conversão de dado for realizada com sucesso; caso contrário, o sistema poderá ficar preso neste ponto, aguardando a conversão.

Outra funcionalidade do sistema é a impressão dos valores lidos, através do conversor AD, no display *Organic* LED (OLED) da placa. Um ponto importante neste caso é que, normalmente, escritas em display são tarefas lentas em relação ao restante do sistema, o que

pode gerar atrasos de modo geral. Por último, tem-se uma função de atraso representada pela função "**Timer0_Wait**", que trava o *Super Loop* durante um tempo de 200ms. Neste ponto, o processador está gastando recursos sem executar nenhuma tarefa, ou seja, todo o sistema está parado aguardando a liberação do processador.

```
while(1)
{
        /* analog input connected to BNC */
        ADC_StartCmd(LPC_ADC,ADC_START_NOW);
        //Wait conversion complete
        while (!(ADC_ChannelGetStatus(LPC_ADC,
                ADC_CHANNEL_5,ADC_DATA_DONE)));
        val = ADC_ChannelGetData(LPC_ADC,ADC_CHANNEL_5);

        /* output values to OLED display */
        intToString(val, buf, 10, 10);
        oled_fillRect((1+6*6),1, 80, 8, OLED_COLOR_WHITE);
        oled_putString((1+6*6),1, buf, OLED_COLOR_BLACK,
                        OLED_COLOR_WHITE);
        /* delay */
        Timer0_Wait(200);
    }
}
```

Figura 5 - Código de referência desenvolvido com base em SL

Para iniciar o processo de migração, o próximo passo é separar essas funcionalidades em tarefas, para que elas sejam gerenciadas pelo escalonador do RTOS, conforme será apresentado a seguir.

5.1 Criando as tarefas

Inicialmente, o foco deve ser a funcionalidade de cada tarefa, deixando suas outras propriedades, como, por exemplo, a prioridade, para serem definidas posteriormente. Primeiro será criada uma tarefa para leitura de dados e outra para a impressão destes no display. Seguindo o protótipo de função para a implementação de uma tarefa no FreeRTOS™, conforme descrito na Seção 4, a tarefa para leitura de dados do conversor AD ficará de acordo com a Figura 6. Outra tarefa será criada para executar a impressão dos dados lidos do conversor analógico digital, no display OLED, conforme descrito na Figura 7.

```
static void vLeituraAD( void *pvParameters )
{
   long lDado;

   for( ;; )
   {
       /* Entrada analógica conectada ao BNC */
       ADC_StartCmd(LPC_ADC,ADC_START_NOW);
       //Aguarda a finalização da conversão
       while (!(ADC_ChannelGetStatus(LPC_ADC,
              ADC_CHANNEL_5,ADC_DATA_DONE)));
       lDado = ADC_ChannelGetData(LPC_ADC,ADC_CHANNEL_5);
   }
}
```

Figura 6 - Tarefa para leitura do conversor AD

```
static void vImprimeDado( void *pvParameters )
{
   long lDado = 0;
   uint8_t buf[10];

   for( ;; )
   {
       /* Valores de saída para o display OLED */
       StringCtrl_intToString(lDado, buf, 10, 10);
       oled_fillRect((1+6*6),1, 80, 8, OLED_COLOR_WHITE);
       oled_putString((1+6*6),1, buf, OLED_COLOR_BLACK,
       OLED_COLOR_WHITE);
   }
}
```

Figura 7 - Tarefa para impressão de dados no display OLED

É importante observar que as tarefas foram criadas, mas ainda não existe nenhum mecanismo de troca de informações ou alguma forma de sincronização entre elas. Neste ponto, portanto, a aplicação está separada em tarefas, e o próximo passo é determinar quais serão as características que envolverão a criação de cada uma delas no sistema, ou seja:

a) qual deve ser a prioridade de cada tarefa;
b) definir se a tarefa deve ser periódica ou não;
c) definir como ocorrerá a troca de dados entre as tarefas.

5.2 Definindo características e prioridades

Supondo que a tarefa de leitura de dados seja uma parte importante do sistema, ou seja, que periodicamente uma leitura de dado seja efetuada, independentemente do que o sistema esteja fazendo, esta tarefa deve ser configurada com uma prioridade maior do que as demais.

Sendo assim, ela precisa, de alguma forma, liberar o processador para as outras tarefas com menor prioridade. Uma forma de isto acontecer é configurando a tarefa para ser periódica. A periodicidade pode respeitar, por exemplo, o tempo de 200ms que era utilizado como atraso na solução com SL. Depois de efetuada a leitura do conversor AD, esta tarefa entrará em modo bloqueado, liberando o processador para as demais. Utilizando-se a função **vTaskDelayUntil** do FreeRTOS™ para bloquear a tarefa por um determinado período de tempo, o código ficará de acordo com a Figura 8.

```
static void vLeituraAD( void *pvParameters )
{
    long lDado;
    portTickType xLastWakeTime = xTaskGetTickCount();

    for( ;; )
    {
        /* Imprime o nome da tarefa. */
        vPrintString( pcTaskName );
        /* Entrada analógica conectada ao BNC */
        ADC_StartCmd(LPC_ADC,ADC_START_NOW);
        //Aguarda finalização da conversão
        while (!(ADC_ChannelGetStatus(LPC_ADC,
               ADC_CHANNEL_5,ADC_DATA_DONE)));
        lDado = ADC_ChannelGetData(LPC_ADC,ADC_CHANNEL_5);

        vTaskDelayUntil( &xLastWakeTime,
                         ( 200 / portTICK_RATE_MS ) );
    }
}
```

Figura 8 - Tarefa periódica para leitura do conversor AD

Já a tarefa de impressão de dados, por ser a mais lenta, pode ser configurada para rodar de forma contínua, mas com prioridade menor que a tarefa de leitura. Dessa forma, a cada 200ms, a tarefa de impressão sofrerá uma preempção do escalonador, que passará os recursos do processador para a tarefa de leitura, conforme ilustrado na Figura 9.

Como a arquitetura do sistema está definida, o próximo passo é criar as tarefas na função principal **main**, de acordo com os padrões do FreeRTOS™. A função **xTaskCreate** deve ser utilizada para a criação de tarefas e precisa receber os seguintes parâmetros:

a) ponteiro para a função que implementa a tarefa;
b) nome para a tarefa (em forma de texto);
c) quanto de memória *stack* será necessário para a tarefa (em *words*);

d) qual parâmetro deve ser passado para essa tarefa todas as vezes que for executada (*NULL* - se não passar nenhum parâmetro);
e) prioridade da tarefa;
f) ponteiro para guardar o *handle* da tarefa, caso necessário (*NULL* - se não utilizar).

O código com a criação das tarefas para a aplicação está representado na Figura 10.

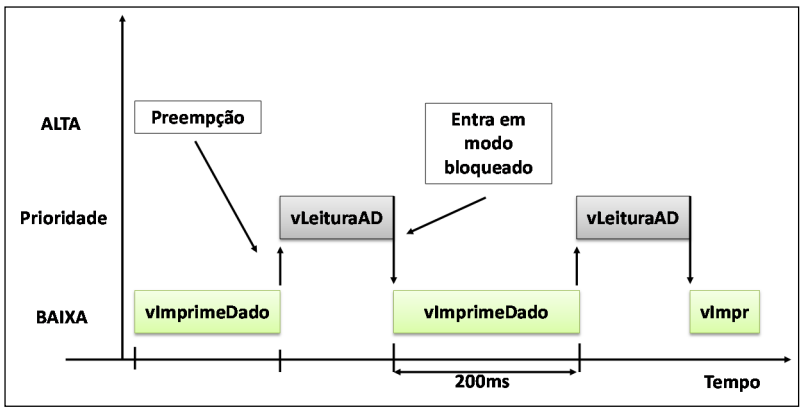

Figura 9 - Diagrama de tempo de execução do sistema

O sistema criado possui duas tarefas independentes, **vLeituraAD** e **vImprimeDado**, que possuem prioridades diferentes. O escalonador é responsável por rodá-las, mesmo se houver algum problema com uma delas, garantindo a integridade da aplicação e evitando a ocorrência de travamento.

```
int main( void )
{
    DriversCtrl_initAll();

    oled_init();
    oled_clearScreen(OLED_COLOR_WHITE);

    xTaskCreate( vLeituraAD, "Leitura do AD", 240,
              NULL, 2, NULL );
    xTaskCreate( vImprimeDado, "Impressao de dados",
              240, NULL, 1, NULL );

    /* Inicia o escalonador. */
    vTaskStartScheduler();

    for( ;; );
    return 0;
}
```

Figura 10 - Código da função main com a criação das tarefas do sistema

Ainda falta resolver uma questão, que é o compartilhamento dos dados lidos na tarefa de leitura do conversor AD com a tarefa de impressão de dados. Este assunto será abordado na próxima seção.

5.3 Compartilhando dados entre tarefas

Para o compartilhamento entre tarefas, o recurso disponível pelo FreeRTOS™ é a utilização de estruturas de filas (*queues*). Pode-se criar uma estrutura de fila que armazena um número máximo de valores e que pode ser acessada por qualquer tarefa do sistema.

A função que cria uma estrutura de fila é a **xQueueCreate**, que recebe como parâmetros o número máximo e o tamanho de elementos que serão armazenados. Ao criar a fila, um ponteiro é retornado com o endereço de acesso à estrutura. O código com a criação de uma fila que suporta até 5 valores de tamanho **long** pode ser visto na Figura 11. É importante observar que o ponteiro para guardar o endereço da estrutura de fila criada deve ser declarado como global, permitindo, assim, que qualquer tarefa possa acessá-la.

```
int main( void )
{
    DriversCtrl_initAll();

    oled_init();
    oled_clearScreen(OLED_COLOR_WHITE);

    xQueue = xQueueCreate( 5, sizeof( long ) );
    if( xQueue != NULL )
    {
        xTaskCreate( vLeituraAD, "Leitura do AD", 240,
                NULL, 2, NULL );
        xTaskCreate( vImprimeDado, "Impressao de
                dados", 240, NULL, 1, NULL );
        /* Inicia o escalonador. */
        vTaskStartScheduler();
    }
    for( ;; );
    return 0;
}
```

Figura 11 - Criação da estrutura de fila na função *main*

Agora, basta preencher a fila com os dados lidos na tarefa de leitura para deixá-los disponíveis para a tarefa de impressão. Para colocar um dado na fila, utiliza-se a função **xQueueSendToBack**, que precisa dos seguintes parâmetros:

a) ponteiro para a estrutura da fila;
b) ponteiro para a variável com o valor a ser inserido;
c) tempo em que a tarefa permanecerá em modo bloqueado, aguardando liberação de espaço na fila.

Logo, a tarefa de leitura de dados **vLeituraAD**, com a inclusão de dados na fila, fica conforme apresentado na Figura 12. Para a tarefa de impressão dos dados, é necessário utilizar uma função para ler os valores que estão na fila. Esta função é a **xQueueReceive**, que recebe os seguintes parâmetros:

a) ponteiro para a estrutura da fila;
b) ponteiro para a variável para receber o valor a ser lido;
c) tempo em que a tarefa permanecerá em modo bloqueado, caso a fila esteja vazia.

```
static void vLeituraAD( void *pvParameters )
{
    long lDado;
    portBASE_TYPE xStatus;
    portTickType xLastWakeTime = xTaskGetTickCount();

    for( ;; )
    {
        /* Entrada analógica conectada ao BNC */
        ADC_StartCmd(LPC_ADC,ADC_START_NOW);
        while (!(ADC_ChannelGetStatus(LPC_ADC,
               ADC_CHANNEL_5,ADC_DATA_DONE)));
        lDado = ADC_ChannelGetData(LPC_ADC,ADC_CHANNEL_5);

        xStatus =xQueueSendToBack( xQueue, &lDado, 0 );
        if( xStatus != pdPASS ) /* A fila está cheia */

        vTaskDelayUntil( &xLastWakeTime, ( 200 /
               portTICK_RATE_MS ) );
    }
}
```

Figura 12 - Tarefa de leitura de dados e inserção na fila

O código da tarefa de impressão **vImprimeDado**, modificada para leitura de dados da fila, pode ser visto na Figura 13. Esta tarefa lê o valor que está na fila e o imprime no display OLED. Caso não haja valores na fila, esta tarefa entra em modo bloqueado até que um valor seja escrito, permanecendo um tempo máximo de 500ms em espera.

```
static void vImprimeDado( void *pvParameters )
{
    long lDado = 0;
    uint8_t buf[10];
    portBASE_TYPE xStatus;
    const portTickType xTicksToWait = 500 / portTICK_RATE_MS;

    oled_clearScreen(OLED_COLOR_WHITE);
    oled_putString(1,1,(uint8_t*)"BNC: ", OLED_COLOR_BLACK,
        OLED_COLOR_WHITE);

    for( ;; )
    {

        xStatus = xQueueReceive( xQueue, &lDado, xTicksToWait);
        if( xStatus == pdPASS )
        {
            /* Valores de saída para o display OLED */
            StringCtrl_intToString(lDado, buf, 10, 10);
            oled_fillRect((1+6*6),1, 80, 8, OLED_COLOR_WHITE);
            oled_putString((1+6*6),1, buf, OLED_COLOR_BLACK,
            OLED_COLOR_WHITE);
        }
    }
}
```

Figura 13 - Tarefa de impressão com leitura de dados da fila

Como a tarefa **vImprimeDado** é executada várias vezes antes de a tarefa **vLeituraAD** ser executada novamente, a fila é esvaziada. Assim, **vImprimeDado** entra em modo bloqueado e sai deste estado toda vez que é escrito um dado na fila, ou seja, a cada 200ms, quando **vLeituraAD** é executada, preenchendo a fila com um novo valor. Logo, ocorre uma sincronização entre as duas tarefas, onde só há impressão de valores quando um novo valor é lido do conversor AD. Mas, se ocorrer algum atraso durante a impressão, o valor será lido de qualquer forma e inserido na fila.

Em alguns momentos, ocorre de ambas as tarefas estarem em modo bloqueado: uma, devido à função **vTaskDelayUntil**, e a outra, por esperar um novo valor ser escrito na fila. Nesta situação, como o escalonador não possui nenhuma tarefa disponível para ser executada, desvia o processador para a tarefa **vApplicationIdleHook**, que pode ser utilizada para as demais funcionalidades do sistema que possuam menor prioridade de aplicação. O diagrama de tempo resultante do sistema está ilustrado na Figura 14.

Finalmente, tem-se a mesma aplicação, concebida inicialmente para um sistema SL, portada para funcionar em um RTOS, que, neste caso, é o FreeRTOS™. Em termos funcionais, a aplicação continua a mesma; porém, agora apresenta características de robustez quanto a travamento de tarefas e confiabilidade quanto à restrição de tempo, no caso da tarefa de leitura **vLeituraAD**.

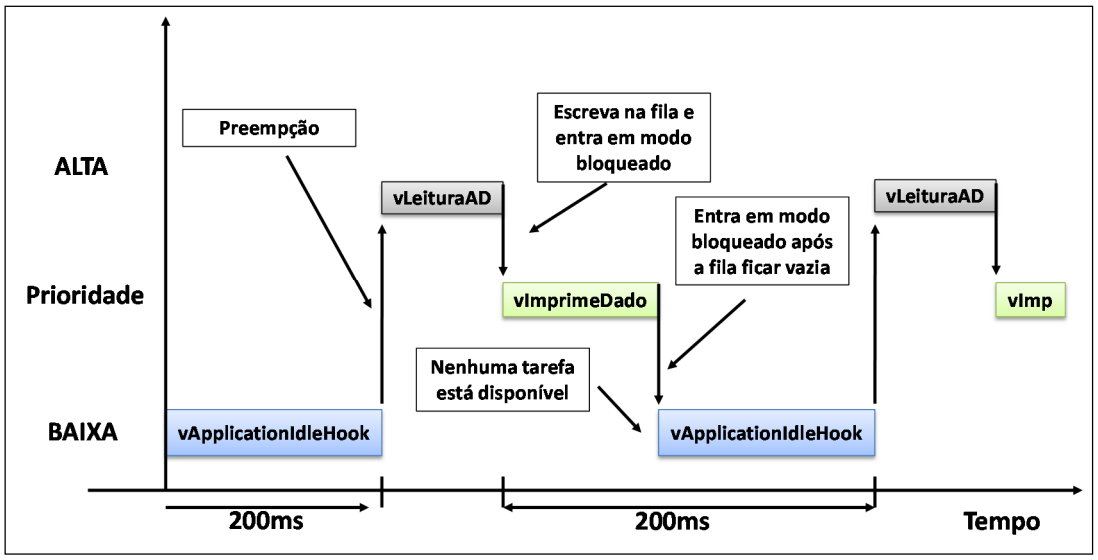

Figura 14 - Diagrama de tempo resultante para o sistema

6 CONCLUSÕES

A escolha do RTOS a ser utilizado depende do desenvolvedor, da arquitetura onde a aplicação será desenvolvida e dos recursos financeiros, dentre outros fatores. Cada sistema operacional implementa seu próprio algoritmo de escalonamento e tem suas próprias funções para a criação de tarefas, filas, sincronização de tarefas etc. Porém, todos têm o mesmo objetivo, que é garantir que o sistema tenha repostas em tempo real e que os recursos sejam compartilhados entre todas as tarefas.

Existem várias formas de se configurar um mesmo sistema, partindo das funcionalidades disponibilizadas pelo RTOS. O resultado apresentado é apenas uma das inúmeras formas possíveis de efetuar a portabilidade de uma aplicação desenvolvida para um sistema SL para um sistema com SO. Com o resultado obtido, foi possível destacar as vantagens da utilização de um sistema operacional em projetos de sistemas embarcados. Apesar de a aplicação utilizada como exemplo ser extremamente simples, pôde-se identificar alguns possíveis pontos de travamento no sistema e a geração de atrasos, sendo assim impossível garantir respostas em tempo real. Isto pode ser observado, por exemplo, no método **vLeituraAD**, onde existe uma espera de conversão de dados pelo conversor AD. Se ocorrer um atraso neste método ou mesmo se houver um travamento, isso não impactará no funcionamento do restante do sistema, que continuará a executar todas as outras tarefas que forem criadas. Outro problema corrigido pelo RTOS foi o de sincronização de tarefas, conforme exemplificado na solução apresentada. Foi possível verificar que a tarefa de impressão de dados no display,

vImprimeDado, só é executada quando há dados disponíveis na fila de mensagens, o que evita ações desnecessárias, tornando o sistema eficiente em termos energéticos.

Qualquer nova necessidade ou funcionalidade que venha a surgir em um projeto com RTOS pode ser facilmente introduzida no sistema. Como estas novas funcionalidades podem ser projetadas como tarefas independentes, basta escrever a solução e escolher com que prioridade ela será executada no sistema, de forma que não interfira no restante do funcionamento e nos requisitos de tempo programados anteriormente.

Portanto, ao portar a aplicação para operar com RTOS, a maioria dos problemas provenientes das arquiteturas mais simples é evitada, tornando-a mais robusta e garantindo que não haja atrasos, principalmente para as tarefas consideradas críticas. Em contrapartida, paga-se um preço por isso, que são os espaços em memória de programa e em memória de dados – ambos necessários para a execução do próprio sistema operacional - além de uma fatia do processador utilizado para sua execução.

REFERÊNCIAS

ANH, T. N. B.; TAN, Su-Lim. Survey and performance evaluation of real-time operating systems (RTOS) for small microcontrollers. **Micro, IEEE**, v. pp, n. 99, p. 1, 2009. Disponível em: <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=5210078&queryText=Survey+and+performance+evaluation+of+real-time++operating+systems+.LB.RTOS.RB.+for+small+microcontrollers>. Acesso em: 28 maio 2013.

BARRY, R. **Using the FreeRTOS Real Time Kernel**: a practical guide. 1. ed. [S.l: s.n.], 2010. p. 195.

DOLINAY, J.; VAŠEK, V.; DOSTÁLEK, P. Utilization of Simple Real-time Operating system on 8-bit microcontroller. **International Journal of Mathematical Models and methods in Applied Sciences**, v. 5, n. 4, p. 789-796, 2011.

HASAN, Al M.; AHMAD, S. Development of a Highly Optimized Preemptive Real Time Operating System (PRTOS). In: INTERNATIONAL CONFERENCE ON, 11., 2008, Khulna. **Anais...** Khulna: IEEE, 2008. p. 25-27.

LAPLANTE, P. A.; OVASKA, S. J. **Real-Time systems design and analysis:** tools for the practitioner. 4. ed. [S.l.]: Wiley-IEEE Press, 2011. p. 584.

MARWEDEL, P. **Embedded systems foundations of cyber-physical systems**. 2. ed. [S.l.]: Springer, 2011. p. 350.

MEDIA. State of embedded market survey. **Embedded System Design Magazine,** 2013.

MOHAMADI, T. Real Time Operating System for AVR Microcontrollers. In: DESIGN & TEST SYMPOSIUM (EWDTS) EAST-WEST, 9., 2011, Sevastopol. **Anais...** Sevastopol: IEEE, set. 2011. p. 376-380. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6116595>. Acesso em: 29 maio 2013.

NAHAS, M. Implementation of highly-predictable time- triggered cooperative scheduler using simple super loop architecture. **International Journal of Electrical & Computer Sciences**, v. 11, p. 33-38, 2011.

SHEN, J. et al. Research of the Real-Time Performance of Operating System. In: WIRELESS COMMUNICATIONS, NETWORKING AND MOBILE COMPUTING (WiCom) INTERNATIONAL CONFERENCE ON, 5., 2009, Beijing. **Anais...** Beijing: IEEE, set. 2009. p. 1-4. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5301855>. Acesso em: 29 maio 2013.

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. **Operating System Concepts**. 9. ed. [S.l.]: John Wiley & Sons, 2012. p. 944.

Recebido em: 19/05/2014

Aprovado em: 05/06/2014